Cell Superscalar (CellSs) User's Manual

Version 1.2

Barcelona Supercomputing Center

May 2007

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Table of Contents

1Introduction....3
2Installation....3
2.1Compilation requirements....3
2.2Compilation....3
2.3Runtime requirements....4
2.4User environment....4
3Programming with CellSs....4
3.1Task selection....4
3.2CellSs syntax....5
4Compiling....7
4.1Usage....7
4.2Examples....8
5Setting the environment and executing....9
5.1Setting the number of SPEs and executing....9
6Programming examples....9
6.1Matrix multiply....9
7Internals Cell Superscalar....12
8Advanced features....13
8.1Using Paraver 13
8.2Configuration file....15
9References....15

1 Introduction

The Cell Broadband Engine (BE) is an heterogeneous multi-core architecture with nine cores. The first generation of the Cell BE includes a 64-bit multithreaded PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high bandwidth Element Interconnect Bus (EIB). The PPE has two levels of on-chip cache and also supports IBM's VMX to accelerate multimedia applications by using VMX SIMD units.

This document is the user manual of the Cell Superscalar framework (CellSs), which is based on a source to source compiler and a runtime library. The supported programming model allows the programmers to write sequential applications and the framework is able to exploit the existing concurrency and to use the different components of the Cell BE (PPE and SPEs) by means of an automatic parallelization at execution time. The requirements we place on the programmer are that the application is composed of coarse grain functions (for example, by applying blocking) and that these functions do not have collateral effects (only local variables and parameters are accessed). These functions are identified by annotations (somehow similar to the OpenMP ones), and the runtime will try to parallelize the execution of the annotated functions (also called tasks).

The source to source compiler separates the annotated functions from the main code and the library provides a manager program to be run in the SPEs that is able to call the annotated code. However, an annotation before a function does not indicate that this is a parallel region (as it does in OpenMP). The annotation just indicates the direction of the parameters (input, output or and inout). To be able to exploit the parallelism, the CellSs runtime takes this information about the parameters and builds a data dependency graph where each node represents an instance of an annotated function and edges between nodes denote data dependencies. From this graph, the runtime is able to schedule for execution independent nodes to different SPEs at the same time. All data transfers required for the computations in the SPEs are automatically performed by the runtime. Techniques imported from the computer architecture area like the data dependency analysis, data renaming and data locality exploitation are applied to increase the performance of the application.

While OpenMP explicitly specifies what is parallel and what is not, with CellSs what is specified are functions whose invocations could be run in parallel, depending on the data dependencies. The runtime will find the data dependencies and will determine, based on them, which functions can be run in parallel with others and which not. Therefore, CellSs provides programmers with a more flexible programming model with an adaptive parallelism level depending on the application input data.

2 Installation

Cell Superscalar is distributed in source code form and must be compiled and installed before using it.

2.1 Compilation requirements

The Cell Superscalar compilation process requires the following system components:

- CBE SDK 1.1 or 2.0
- GNU make
- Optional: automake, autoconf, libtool, bison, flex

2.2 Compilation

To compile and install Cell Superscalar please follow the following steps:

1. Decompres the source tarball

   ```
   $ tar -xvzf CellSS-1.0.tar.gz
   ```

2. Enter into the source directory

   ```
   cd CellSS-1.0
   ```

3. If necessary, check that you have set the PATH and LD_LIBRARY_PATH environment variables to point to the CBE SDK installation.
4. Run the configure script, specifying the installation directory as the *prefix* argument and optionally the CBE SDK installation path in the *with-cellsdk* argument. More information can be obtained by running *./configure --help*.

   ```
   ./configure --prefix=/opt/CellSS
   ```

5. Run make

   ```
   make
   ```

6. Run make install

   ```
   make install
   ```

2.3 Runtime requirements

The Cell Superscalar runtime requires the following system components:

- CBE SDK 1.1 or 2.0

2.4 User environment

If the CBE SDK resides on a non standard directory, then the user must set the PATH and LD_LIBRARY_PATH accordingly.

If Cell Superscalar has not been installed into a system directory, then the user must set the following environment variables:

1. The *PATH* environment variable must contain the *bin* subdirectory of the installation.

   ```
   export PATH=$PATH:/opt/CellSS/bin
   ```

2. The *LD_LIBRARY_PATH* environment variable must contain the *lib* subdirectory from the installation.

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/CellSS/lib
   ```

3 Programming with CellSs

Cell Superscalar applications are based on the parallelization at task level of sequential applications. The tasks (functions or subroutines) selected by the programmer will be executed in the SPE processors. Furthermore, the runtime detects when tasks are data independent between them and is able to schedule the simultaneous execution of several of them on different SPEs. Since, the SPE cannot access the main memory, the data required for the computation in the SPE is transferred by DMA. All the above mentioned actions (data dependency analysis, scheduling and data transfer) are performed transparently to the programmer. However, to take benefit of this automation, the computations to be executed in the Cell BE should be of certain granularity (about 80 µsecs). A limitation on the tasks is that they can only access their parameters and local variables. In case global variables are accessed the compilation will fail.

3.1 Task selection

In the current version of Cell Superscalar it is a responsibility of the application programmer to select tasks of a certain granularity. For example, blocking is a technique that can be applied to increase the granularity of the tasks in applications that operate on matrices. Below there is a sample code for a block matrix multiplication:

```
void block_addmultiply(double C[BS][BS], double A[BS][BS], double B[BS][BS]) {
int i, j, k;

for (i=0; i < BS; i++)
    for (j=0; j < BS; j++)
        for (k=0; k < BS; k++)
            C[i][j] += A[i][k] * B[k][j];
}
```

3.2 CellSs syntax

Starting and finishing CellSs applications

The following optional pragmas indicate the scope of the program that will use the CellSs features.

```
#pragma css start
#pragma css finish
```

When the start pragma is reached all the SPU threads are initiated and run until the finish pragma is reached. Annotated functions have to be called between this two pragmas. If they are not present in the user code, the compiler will automatically insert the start pragma at the beginning of the application and the finish pragma at the end.

Specifying a task

Notation:

```
#pragma css task [ input(<input parameters>) ]opt [ inout(<inout parameters>) ]opt
        [ output(<output parameters>) ]opt [ highpriority ]opt
{ function declaration | function definition }
```

Input clause

List of parameters whose input value will be read.

Inout clause

List of parameters that will be read and writen by the task.

Output clause

List of parameters that will be written to.

Highpriority clause

Specifies that the task will be sent for execution earlier than tasks without the highpriority clause.

Parameter notation:

```
<parameter> [ [<dimension>] ]*
```

Examples:
In this example, the “factorial” task has a single input parameter “n” and a single output parameter “result”.

```
#pragma css task input(n) output(result)
void factorial(unsigned int n, unsigned int *result) {
        *result = 1;
        for (; n > 1; n--) {
                *result = *result * n;
        }
}
```

The next example, has two input vectors “left”, of size “leftSize”, and “right”, of size “rightSize”; and a single output “result” of size “leftSize+rightSize”.

```
#pragma css task  input(left[leftSize], right[rightSize])  output(result[leftSize+rightSize])
void merge(float *left, unsigned int leftSize, float *right, unsigned int rightSize,
        float *result) {
...
}
```

The next example shows another feature. In this case, with the keyword “highpriority” the user is giving hints to the scheduler: the *lu0* tasks will be, when data dependencies allow it, executed before the ones that are not marked as high-priority.

```
#pragma css task highpriority inout(diag)
void lu0(float diag[64][64]){
...
}
```

Waiting on data

Notation:

#pragma css **wait on**(<*list of expressions*>)

On clause

Comma separated list of expressions corresponding to the addresses that the system will wait for.

In Example 1 the vector “data” is generated by bubblesort. The wait pragma waits for this function to finish before printing the result.

Example 1:

```
#pragma css task inout(data[size]) input(size)
void bubblesort(float *data, unsigned int size) {
        ...
}

void main() {
        ...
        bubblesort(data, size);
#pragma css wait on(data)
        for (unsigned int i=0; i < size; i++) {
                printf("%f ", data[i]);
        }
}
```

In Example 2, matrix[N][N] is a 2-dimension array of pointers to 2-dimension arrays of floats. Each of this 2-dimension arrays of floats are generated in the application from annotated functions. The pragma waits on the address to each of these blocks before printing the result in a file.

Example 2:

```
void write_matrix (FILE * file, matrix_t matrix)
{
int rows, columns;
int i, j, ii, jj;

fprintf (file, "%d\n %d\n", N * BSIZE, N * BSIZE);
for (i = 0; i < N; i++)
         for (ii = 0; ii < BSIZE; ii++)
        {
                for (j = 0; j < N; j++){
  #pragma css wait on(matrix[i][j])
                        for (jj = 0; jj < BSIZE; jj++)
                                fprintf (file, "%f ", matrix[i][j][ii][jj]);
                }
                fprintf (file, "\n");
        }
}
```

4 Compiling

All steps of the CellSs compiler have been integrated in a single step compilation, called through `mcc` and the corresponding compilation options, which are indicated in the usage section below. The `mcc` compilation process consists in preprocessing the CellSs pragmas, compilating both for the PPE and SPE with the corresponding compilers (ppu-c99 and spu-c99), embedding the SPE code in the PPE binary (ppu32-embedspu) and linking with the needed libraries (including the CellSs libraries).

The current version is only able to compile single source code applications. A way of overcoming this limitation is to provide through libraries the code that does not contain annotations and that is not calling to annotated functions.

4.1 Usage

The mcc compiler has been designed to mimic the options and behaviour of common C compilers. However, it uses two other compilers internally that may require different sets of compilation options.

To cope with this distinction, there are general options and target specific options. While the general options are applied to PPE code and SPE code, the target specific options allow to specify options to pass to the PPE compiler and the SPE compiler independently.

The list of supported options is the following:

```
> mcc -help
-Dmacro              Option passed to the preprocessors
-g                   Option passed to the native compilers
-g3                  Option passed to the native compilers
-h/-help             Prints this information
-Idir                Option passed to the native preprocessors
-k/-keep             Keep temporary files
-llibrary            Option passed to the PPU compiler
-Ldir                Option passed to the PPU compiler
-noincludes          Don't try to regenerate include directives
-O0                  Option passed to the native compilers
-O1                  Option passed to the native compilers
-O2                  Option passed to the native compilers
-O3                  Option passed to the native compilers
-ofile               Sets the name of the output file
-t/-tracing          Enable program tracing
-v/-verbose          Enables some informational messages
-WPPUc,OPTIONS       Comma separated list of options passed to the PPU compiler
-WPPUl,OPTIONS       Comma separated list of options passed to the PPU linker
-WPPUp,OPTIONS       Comma separated list of options passed to the PPU
preprocessor
-WSPUc,OPTIONS       Comma separated list of options passed to the SPU compiler
-WSPUl,OPTIONS       Comma separated list of options passed to the SPU linker
-WSPUp,OPTIONS       Comma separated list of options passed to the SPU
preprocessor
```

4.2 Examples

```
> mcc -O3 matmul.c -o matmul
```

Compilation of application file `matmul.c` with -O3 optimization level. If there are no compilation errors, the executable file “matmul” is created which can be called from the command line (“`> ./matmul ...`”).

```
> mcc -keep cholesky.c -o cholesky
```

Compilation with -keep option of cholesky.c application. Option -keep will not delete the intermediate files (files generated by the preprocessor, object files, ...). If there are no compilation errors, the executable file “cholesky” is created.

```
> mcc -O2 -t matmul.c -o matmul
```

Compilation with the -t (tracing) feature. When executing “matmul”, a tracefile of the execution of the application will be generated.

```
> mcc -O2 -WSPUc,-funroll-loops,-ftree-vectorize,-ftree-vectorizer-verbose=3
matmul.c -o matmul
```

The list of flags after the -WSPUc are passed to the SPU compiler (for example, spu-c99). These options perform automatic vectorization of the code to be run in the SPEs. Note: vectorization seems to not properly work with -O3.

5 Setting the environment and executing

5.1 Setting the number of SPEs and executing

Before executing a Cell Superscalar application, the number of SPE processors to be used in the execution have to be defined. The default value is 8, but it can be set to a different number with the CSS_NUM_SPUS environment variable, for example:

```
> export CSS_NUM_SPUS=6
```

Cell Superscalar applications are started from the command line in the same way as any other application. For example, for the compilation examples of section 4.2, the applications can be started as follow:

```
> ./matmul <pars>
> ./cholesky <pars>
```

6 Programming examples

This section presents a programming example for the block matrix multiplication. The code is not complete, but you can find the complete and working code in the directory <install_dir>/share/docs/cellss/examples/, being <install_dir> the installation directory. More examples are provided in this directory also.

6.1 Matrix multiply

This example presents a CellSs code for a block matrix multiply. The block size is of 64 x 64 floats.

```
#pragma css task input(A, B) inout(C)
```

```
static void block_addmultiply(float C[BS][BS], float A[BS][BS], float B[BS][BS]) {
int i, j, k;

for (i=0; i < BS; i++)
        for (j=0; j < BS; j++)
                for (k=0; k < BS; k++)
                        C[i][j] += A[i][k] * B[k][j];
}

int main(int argc, char **argv) {
int i, j, k;

initialize(argc, argv,  A, B, C);
#pragma css start
for (i=0; i < N; i++)
        for (j=0; j < N; j++)
                for (k=0; k < N; k++)
                        block_addmultiply( C[i][j], A[i][k], B[k][j]);
...
}
#pragma css finish
```

This main code will run in the Cell PPE and the block_addmultiply calls will be executed in the SPE processors. It is important to note that the sequential code (including the annotations) can be compiled and run in a sequential processor. This is very useful for debugging the algorithms.

However, the code is not vectorized, and if a compiler that does not vectorize the code is used, it is not going to be very efficient. The programmer can pass to the corresponding compiler the compilation flags that autovectorize the SPE (see section 4.2). Another option will be to manually provide a vectorized code as the one that follows:

```
#ifdef SPU_CODE
#include <spu_intrinsics.h>
#endif
#define BS 64
#define BSIZE_V BS/4
#pragma css task input(A, B) inout(C)
void block_addmultiply(float C[BS][BS], float A[BS][BS], float B[BS][BS])
{
  vector float *Bv=(vector float*)B;
  vector float *Cv=(vector float*)C;
  vector float elem;
```

```
 int i, j, k;

 for (i=0; i<BS; i++) {
  for (j=0; j<BS; j++) {
        elem = spu_splats (A[i*BS+j]);
        for (k=0; k<BSIZE_V; k++) {
              Cv[i*BSIZE_V+k] = spu_madd (elem, Bv[j*BSIZE_V+k], Cv[i*BSIZE_V+k]);
        }
  }
 }
 }
```

This code can even be improved by unrolling the inner loop (this is automatically done when option -O3 is used). Even more, the code can be improved if the data is prefetched in advance, as the next version of the sample code does:

```
#ifdef SPU_CODE
#include <spu_intrinsics.h>
#endif
#define BS 64
#define BSIZE_V BS/4
#pragma css task input(A, B) inout(C)
void
matmul (float A[BSIZE][BSIZE], float B[BSIZE][BSIZE], float C[BSIZE][BSIZE])
{
vector float *Bv = (vector float *) B;
vector float *Cv = (vector float *) C;

vector float elem;
int i, j;
int i_size;
int j_size;
vector float tempB0, tempB1, tempB2, tempB3;

i_size=0;
for (i = 0; i < BSIZE; i++)
{
        j_size=0;
        for (j = 0; j < BSIZE; j++)
        {
              elem = spu_splats (A[i][j]);
```

```
                tempB0 = Bv[j_size+0];
                tempB1 = Bv[j_size+1];
                tempB2 = Bv[j_size+2];
                Cv[i_size+0] = spu_madd (elem, tempB0, Cv[i_size+0]);
                tempB3 = Bv[j_size+3];
                Cv[i_size+1] = spu_madd (elem, tempB1, Cv[i_size+1]);
                tempB0 =  Bv[j_size+4];
                Cv[i_size+2] = spu_madd (elem, tempB2, Cv[i_size+2]);
                tempB1 =  Bv[j_size+5];
                Cv[i_size+3] = spu_madd (elem, tempB3, Cv[i_size+3]);
                tempB2 =  Bv[j_size+6];
                Cv[i_size+4] = spu_madd (elem, tempB0, Cv[i_size+4]);
                tempB3 =  Bv[j_size+7];
                Cv[i_size+5] = spu_madd (elem, tempB1, Cv[i_size+5]);
                tempB0 =  Bv[j_size+8];
                Cv[i_size+6] = spu_madd (elem, tempB2, Cv[i_size+6]);
                tempB1 =  Bv[j_size+9];
                Cv[i_size+7] = spu_madd (elem, tempB3, Cv[i_size+7]);
                tempB2 =  Bv[j_size+10];
                Cv[i_size+8] = spu_madd (elem, tempB0, Cv[i_size+8]);
                tempB3 =  Bv[j_size+11];
                Cv[i_size+9] = spu_madd (elem, tempB1, Cv[i_size+9]);
                tempB0 =  Bv[j_size+12];
                Cv[i_size+10] = spu_madd (elem, tempB2, Cv[i_size+10]);
                tempB1 =  Bv[j_size+13];
                Cv[i_size+11] = spu_madd (elem, tempB3, Cv[i_size+11]);
                tempB2 =  Bv[j_size+14];
                Cv[i_size+12] = spu_madd (elem, tempB0, Cv[i_size+12]);
                tempB3 =  Bv[j_size+15];
                Cv[i_size+13] = spu_madd (elem, tempB1, Cv[i_size+13]);
                Cv[i_size+14] = spu_madd (elem, tempB2, Cv[i_size+14]);
                Cv[i_size+15] = spu_madd (elem, tempB3, Cv[i_size+15]);
                j_size += BSIZE_V;
        }
        i_size += BSIZE_V;
}
}
```

7 Internals Cell Superscalar

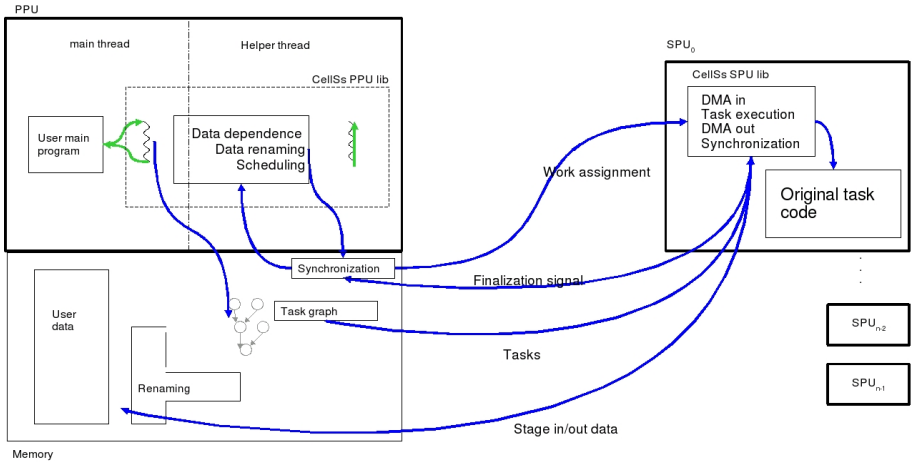

Figure 1: Cell Superscalar runtime behavior

When compiling a CellSs application with mcc the resulting object files are linked with the CellSs runtime library. Then, when the application is started, the CellSs runtime is automatically invoked. The CellSs runtime is decoupled in two parts: one runs in the PPU and the other in each of the SPUs. In the PPU, we will differentiate between the master thread and the helper thread.

The most important change in the original user code is that the CellSs compiler replaces the calls to the css_addTask function whenever a call to an annotated function appears. At runtime, these calls to the css_addTask function will be responsible for the intended behavior of the application in the Cell BE processor. At each call to css_addTask, the master thread will do the following actions:

- A node that represents the called task in a task graph is added.
- Data dependency analysis of the new task with other previously called tasks.
- Parameters renaming: similarly to register renaming, a technique from the superscalar processor area, we do renaming of the output and input/output parameters. For every function call that has a parameter that will be written, instead of writing to the original parameter location, a new memory location will be used, that is, a new instance of that parameter will be created and it will replace the original one, becoming a renaming of the original parameter location. This allows to execute that function call independently from any previous function call that would write or read that parameter. This technique allows to effectively remove some data dependencies by using additional storage, and thus improving the chances to extract more parallelism.

The helper thread is the one that decides when a task should be executed and also monitors the execution of the tasks in the SPUs.

Given a task graph, the helper thread schedules tasks for execution in the SPUs. This scheduling follows some guidelines:

- A task can be scheduled if its predecessor tasks in the graph have finished their execution.
- To reduce the overhead of the DMA, groups of tasks are submitted to the same SPU.
- Data locality is exploited by keeping task outputs in the SPU local memory and scheduling tasks that reuse this data to the same SPU.

The helper thread synchronizes and communicates with the SPUs using a specific area of the PPU main memory for each SPU. The helper thread indicates the length of the group of tasks to

be executed and information related to the input and output data of the tasks.

The SPUs execute a loop waiting for tasks to be executed. Whenever a group of tasks is submitted for execution, the SPU starts the DMA of the input data, processes the tasks and writes back the results to the PPU memory. The SPU synchronizes with the PPU to indicate end of the group of tasks using a specific area of the PPU main memory.

8 Advanced features

8.1 Using Paraver

To understand the behavior and performance of the applications, the user can generate Paraver tracefiles of their Cell Superscalar applications.

If the -t/-tracing flag is enabled at compilation time, the application will generate a Paraver tracefile of the execution. The default name for the tracefile is “gss-trace-*id*.prv”. The name can be changed by setting the environment variable CSS_TRACE_FILENAME. For example, if it is set as follows:

```
> export CSS_TRACE_FILENAME=tracefile
```

after the execution, the files: tracefile-0001.row, tracefile-0001.prv and tracefile-0001.pcf are generated. All these files are required by the Paraver tool.

The traces generated by Cell Superscalar can be visualized and analyzed with Paraver. Paraver is distributed independently of Cell Superscalar. Paraver can be obtained from:

http://www.cepba.upc.es/paraver/

Several configuration files to visualise and analyse Cell superscalar tracefiles are provided in the Cell Superscalar distribution in the directory <install_dir>/share/cellss/paraver_cfgs/.

The following table summarizes what is shown by each configuration file:

Configuration file	Feature shown
DMA_bw.cfg	DMA (in+out) bandwidth per SPU
DMA_bytes.cfg	Bytes being DMAed (in+out) by each SPU
execution_phases.cfg	Profile of percentage of time spent by each thread (master, helper) and SPE at each of the major phases in the run time library (ie. generating tasks, scheduling, DMA, task execution,...)
flushing.cfg	Intervals (dark blue) where each SPU is flushing its local trace buffer to main memory. For the main and helper thread the flushing is actually to disk. Overhead in this case is thus significant as this stalls the respective engine (task generation or submission)
general.cfg	Several views : Run time phase, task id, task duration, task number, transfer direction, transfer bandwidth
stage_in_out_phase.cfg	Identification of DMA in (grey) and out phases (green)
task.cfg	Outlined function being executed by each SPE
task_number.cfg	Number (in order of task generation) of task being executed by each SPE.

	Light green for the initial tasks in program order, blue for the last tasks in program order. Intermixed green and blue indicate out of order execution.
Task_profile.cfg	Time (microseconds) each SPE spent executing the different task. Change statistic to: - #burst: number of tasks of each type by the SPEs. - Average burst time: Avg. duration of task type
Total_DMA_bw.cfg	Total DMA (in+out) bandwidth to Memory
2dh_inbw.cfg	Histogram of the bandwidth achieved by individual DMA IN transfers. Zero on the left, 10GB/s on the right. Darker == more times a transfer at such bandwidth occurred.
2dh_inbytes.cfg	Histogram of bytes read by the stage in DMA transfers
2dh_outbw.cfg	Histogram of the bandwidth achieved by individual DMA OUT transfers. Zero on the left, 10GB/s on the right. Darker == more times a transfer at such bandwidth occurred.
2dh_outbytes.cfg	Histogram of bytes written by the stage out DMA transfers
3dh_duration_tasks.cfg	Histogram of duration of SPE tasks. One plane per task (Fixed Value Selector) Left column: 0 microseconds, right column 3000 ms. Darker: higher number of instances of that duration.

8.2 Configuration file

With the objective of tuning the behaviour of the CellSs runtime, a configuration file where some variables are set is introduced. However, we do not recommend to play with this variables unless the user considers that it is required to improve the performance of her/his applications. The current set of variables is the following (the value between parenthesis denote the default value):

- scheduler.min_tasks (16): defines minimum number of generated tasks before they are scheduled (no more tasks are scheduled while this number is not reached).
- scheduler.initial_tasks (128): defines the number of generated tasks at the beginning of the execution of an application before starting the scheduling of their execution in the SPEs.
- scheduler.max_strand_size(8): defines the maximum number of tasks that are simultaneously scheduled to an SPE.
- scheduler.min_strand_size (6): defines de minimum number of tasks that are

simultaneously scheduled to an SPE.

- task_graph.task_count_high_mark(1000): defines the maximum number of non executed tasks that the graph will hold. The purpose of this variable is to control the memory usage.
- task_graph.task_count_low_mark(900): whenever the task graph reaches task_graph.task_count_high_mark tasks, the task graph generation is suspended until the number of non executed tasks goes below task_graph.task_count_low_mark.

This variables are set in a plain text file, with the following syntax:

scheduler.min_tasks = 32

scheduler.initial_tasks = 128

scheduler.max_strand_size = 8

scheduler.min_strand_size = 4

task_graph.task_count_high_mark = 2000

task_graph.task_count_low_mark = 1500

The file where the variables are set is indicated by setting the CSS_CONFIG_FILE environment variable. For example, if the file “file.cfg” contains the above variable settings, the following command can be used:

```
> export CSS_CONFIG_FILE=file.cfg
```

A sample configuration file for the execution of Cell Superscalar applications is located in <install_dir>/share/docs/cellss/examples/HM_transpose.cfg.

9 References

1. Cell Superscalar website, www.bsc.es/cellsuperscalar
2. Pieter Bellens, Josep M. Perez, Rosa M. Badia and Jesus Labarta. CellSs: A Programming Model for the Cell BE Architecture, *in proceedings of the ACM/IEEE SC 2006 Conference*, November 2006.
3. Paraver website, www.cepba.upc.edu/paraver